PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6: G06F 9/45, 9/44 — A1

(11) International Publication Number: **WO 98/02808**

(43) International Publication Date: 22 January 1998 (22.01.98)

(21) International Application Number: PCT/US97/11891

(22) International Filing Date: 10 July 1997 (10.07.97)

(30) Priority Data:
08/680,270 11 July 1996 (11.07.96) US

(71) Applicant: TANDEM COMPUTERS INCORPORATED [US/US]; 10435 N. Tantau Avenue, Loc. 200-16, Cupertino, CA 95014 (US).

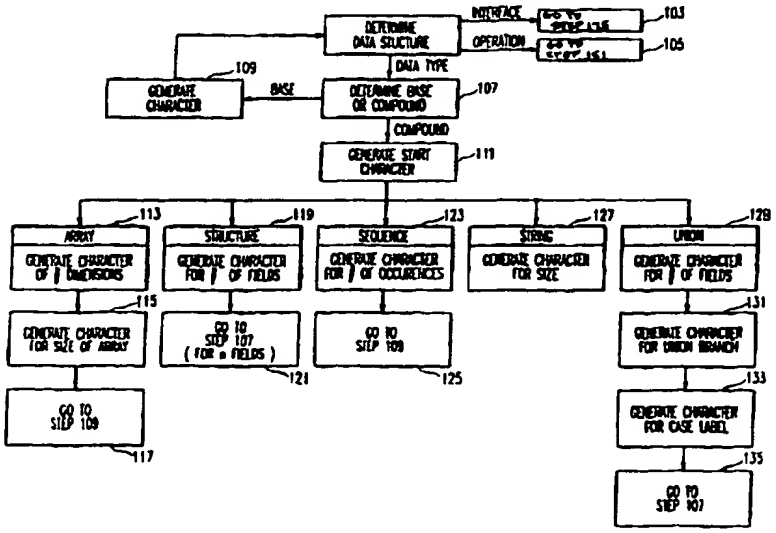
(72) Inventor: SCHOFIELD, Andrew; Lindenbuehl 27, CH-6330 Cham (CH).

(74) Agents: GRANATELLI, Lawrence, W. et al.; Graham & James LLP, 600 Hansen Way, Palo Alto, CA 94304 (US).

(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND APPARATUS FOR DESCRIBING AN INTERFACE DEFINITION LANGUAGE-DEFINED INTERFACE, OPERATION, AND DATA TYPE



(57) Abstract

A method for defining Interface Definition Language-defined data types, operations, or interfaces is defined. In particular, an ASCII string descriptor is generated that identifies the data type, interface, or operation. Characters are used to identify the base types of any data types, while a combination of characters and numerals are used to identify compound types and characteristics of those compound types. For operations, characters and numerals are used to identify the number and types of parameters, number of exceptions, and number of contexts contained in the operation. Interfaces are described using a numeral that indicates the number of operations included in the interfaces. By converting the original IDL description to an ASCII string, generic functions can be created to transport object calls across heterogeneous systems.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

METHOD AND APPARATUS FOR DESCRIBING AN INTERFACE DEFINITION LANGUAGE-DEFINED INTERFACE, OPERATION, AND DATA TYPE

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for describing an interface definition language-defined interface, operation, and data type. In particular, data types are described using a series of ASCII characters.

2. Background

Distributed object computing combines the concepts of distributed computing and object-oriented computing. Distributed computing consists of two or more pieces of software sharing information with each other. These two pieces of software could be running on the same computer or on different computers connected to a common network. Most distributed computing is based on a client/server model. With the client/server model, two major types of software are used: client software, which requests the information or service, and server software, which provides or implements the information or service.

Object-oriented computing is based upon the object model where pieces of code called "objects"--often abstracted from real objects in the real world--contain data and may have actions (also known as "operations") performed on them. An object is defined by its interface (or "class" in C++ parlance). The interface defines the characteristics and behavior of a kind of object, including the operations that can be performed on objects of that interface and the parameters to that operation. A specific instance of an object is identified within a distributed object system by a unique identifier called an object reference.

In a distributed object system, a client application sends a request (or "invocation" or "object call") to a server application containing an indication of the operation to be performed on a specific object, the parameters to that operation, the object reference for that object, and a mechanism to return error information (or "exception information") about the success or failure of a request. The server application receives the request and carries out the request via a server "implementation." The implementation satisfies the client's request for an operation on

a specific object. The implementation includes one or more methods, which are the portions of code in the server application that actually do the work requested of the implementation. If the implementation is carried out successfully, the server application returns a response to the client, if necessary. The server application may also return exception information.

To standardize distributed object systems, the Object Management Group ("OMG"), a consortium of computer software companies, proposed the Common Object Request Broker Architecture ("CORBA"). Under the CORBA standard, an Object Request Broker ("ORB") provides a communication hub for all objects in the system passing the request to the server and returning the response to the client. Commercial ORB's are known in the art and a common type is IBM's System Object Model ("SOM"). On the client side, the ORB handles requests for the invocation of a method and the related selection of servers and methods. When an application sends a request to the ORB for a method to be performed on an object, the ORB validates the arguments contained in the request against the interface for that object and dispatches the request to the server, starting it if necessary. On the server side, the ORB uses information in the request to determine the best implementation to satisfy the request. This information includes the operation the client is requesting, what type of object the operation is being performed on, and any additional information stored for the request. In addition, the server-side ORB validates each request and its arguments. The ORB is also responsible for transmitting the response back to the client.

Both the client application and the server application must have information about the available interfaces, including the objects and operations that can be performed on those objects. To facilitate the common sharing of interface definitions, OMG proposed the Interface Definition Language ("IDL"). IDL is a definition language (not a programming language) that is used to describe an object's interface; that is, the characteristics and behavior of a kind of object, including the operations that can be performed on those objects.

IDL interfaces define a set of operations that a client can invoke on an object. An interface can declare one or more exceptions, which indicate that an IDL operation did not perform successfully. Operations may receive parameters and return a return

value. Each parameter to an operation may have a "direction" that indicates whether the value is passed from client to server ("in"), from server to client ("out"), or in both directions ("inout"). The parameter also has a data type that constrains its possible values. Operations may also optionally have a "one-way" attribute, which specifies which invocation semantics the communication service must provide for invocations of a particular operation. When a client invokes an operation with the one-way attribute, the invocation semantics are "best-effort", implying that the operation will be implemented by the server at most once. If an attempt to implement the operation fails, the server does not attempt to implement the operation again. An operation with the one-way attribute must specify a void return type and must not contain any output parameters.

Data types are used to describe the accepted values of IDL operation parameters, exceptions, and return values. IDL supports two categories of data types: basic and compound. Basic types include short integers, long integers, long long integers, unsigned long integers, unsigned short integers, floating points, double, character, boolean, and octet. Compound types include enum, string, struct, array, union, sequence, and "any" types. The struct type is similar to a C structure; it lets interface designers create a complex data type using one or more type definitions. The sequence type lets interface designers pass a variable-size array of objects. The "any" type can represent any possible data type--basic or compound.

IDL is designed to be used in distributed object systems implementing OMG's ("CORBA"), Revision 2.0 specification. In a typical system implementing the CORBA specification, interface definitions are written in an IDL-defined source file (also known as a "translation unit"). The source file is compiled by an IDL compiler that generates programming-language-specific files, including client stub files, server stub files, and header files. Client stub files are language-specific mappings of IDL operation definitions for an object type into procedural routines, one for each operation. When compiled by a language-specific compiler and linked into a client application, the stub routines may be called by the client application to invoke a request. Similarly, the server stub files are language-specific mappings of IDL operation definitions for an object type (defined by an interface) into procedural routines. When compiled and linked into a server application, the server application can call these routines when a

corresponding request arrives. Header files are compiled and linked into client and server applications and are used to define common data types.

One drawback to using strict IDL-defined operations within an application is the inability to use these operations in generic functions. Generic functions can be utilized across heterogeneous platforms and, therefore, promote code reuse across these varying platforms. In the prior art, however, each application must call a specific client stub function to access an operation contained in a particular interface. As additional operations are added to the interface, new stub functions must be generated. Thus, the application can become more and more complex over time.

Another drawback to strict IDL-defined operations is the inability to compare interfaces to determine compatibility. In certain instances, a server may not be able to provide a requested service because the server does not accommodate that interface and its associated operations. The server may, however, have a compatible interface that provides a similar operation. Currently, there is no method available for determining compatibility of interfaces and/or operations.

Accordingly, there is a need for using IDL-defined interfaces, operations, and parameters in generic functions.

Additionally, there is a need for facilitating the comparison of IDL-defined interfaces, operations, and data types.

SUMMARY OF THE INVENTION

The present invention is directed to a method and system and computer program product that satisfies the need for using IDL-defined interfaces, operations, and data types in generic functions. In addition, the method of the present invention satisfies the need for facilitating the comparison of IDL-defined interfaces, operations, and parameters. The method includes the step of generating a new ASCII string descriptor that identifies the interface, operation, or parameter. More particularly, characters are used to identify the base type of any parameter, while a combination of characters and numerals are used to identify compound type parameters and characteristics of those compound type parameters. For operations, characters and numerals are used to identify the number and types of parameters, number of exceptions, and number of contexts contained in the operation. Interfaces are

described using a numeral that indicates the number of operations included in the interfaces.

By converting the original IDL description to an ASCII string, generic functions can be created to transport object calls across heterogeneous systems. Moreover, code generators can simply emit strings to fully describe an interface rather than a complex link of data structures. In addition, individual operations and data types can easily be described using the method of the present invention.

A more complete understanding of the method for describing data types, operations, and interfaces will be afforded to those skilled in the art, as well as a realization of additional advantages and objects thereof, by a consideration of the following detailed description of the preferred embodiment. Reference will be made to the appended sheets of drawings which will first be described briefly.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a diagram of a client/server system using the method of the present invention.

Figs. 2a and 2b are diagrams of alternative configurations for Common Execution Environment capsules.

Fig. 3 is a diagram of a Common Execution Environment capsule and its core components.

Fig. 4 is a diagram of the compilation of IDL source files.

Fig. 5 is a flow chart depicting the generation of a CIN descriptor for base and compound data types.

Fig. 6 is a flow chart depicting the generation of a CIN descriptor for an operation.

Fig. 7 is a flow chart depicting the generation of a CIN descriptor for an interface.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to

the same or like parts.

I. System Overview

As illustrated in Figure 1, the method of the present invention is designed for use in a distributed (client/server) computing environment 10. The client and server systems are connected by network connections 12, such as internet connections or the connections of a local area network. A server computer 11 communicates over a bus or I/O channel 20 with an associated disk storage subsystem 13. The server system 11 includes a CPU 15 and a memory 17 for storing current state information about program execution. A portion of the memory 17 is dedicated to storing the states and variables associated with each function of the program which is currently executing on the client computer. The client computer 21 similarly includes a CPU 27 and associated memory 23, and an input device 29, such as a keyboard 29, or a mouse 31 and display device 33, such as a video display terminal ("VDT"). The client CPU communicates over a bus or I/O channel 40 with a disk storage subsystem 33 and via I/O channel 41 with the keyboard 29, VDT 33 and mouse 31. Both computers are capable of reading various types of media, including floppy disks and CD-ROMs.

The client memory 27 includes a client application 77 and client stubs 79 loaded therein. Similarly, the server memory 17 includes a server application 87 and server stubs 89. In addition, both the client memory and the server memory include an execution environment ("CEE") 75, 85 (as discussed below).

The client/server model as shown in Figure 1 is merely demonstrative of a typical client/server system. Within the context of the present invention, the "client" is an application that requests that operations be performed on an object while the "server" is an application that implements the operation on the object. Indeed, both the client and server application may reside on the same computer and within a common capsule, as discussed below. Most likely, however, the client and server application will reside on separate computers using different operating systems. The method of the present invention will be discussed with reference to two capsules running on separate machines.

II. Distributed Computing Environment

The method and apparatus of the present invention may be utilized within any distributed computing environment. In a preferred embodiment, the Common Execution Environment ("CEE") 75, 85, which is a component of the Tandem Message Switching Facility ("MSF") Architecture, is used. The CEE activates and deactivates objects and is used to pass messages between client and server applications loaded in CEE capsules. The CEE may be stored in the memory of a single machine. More likely, however, the CEE and client and server applications will be loaded on multiple machines across a network as shown in Figure 1. The client-side CEE 75 is stored in the client memory 23. The server-side CEE 85 is stored in server memory 17.

The CEE uses a "capsule" infrastructure. A capsule encapsulates memory space and one or more execution streams. A capsule may be implemented differently on different systems depending upon the operating system used by the system. For instance, on certain systems, a capsule may be implemented as a process. On other systems, the capsule may be implemented as a thread. Moreover, client and server applications may be configured within different capsules contained on different machines as shown in Figure 1. Alternatively, the different capsules may be configured as shown in Figure 2. Figure 2a shows a client application 77 loaded in a single capsule 81 and a server application 87 may be loaded in a separate capsule 85. Both capsules, however, are stored on the same machine 21. Both the client and server applications may also be loaded within a single capsule 81 on the same machine 21 as shown in Figure 2b. As stated above, the method of the present invention will be described with reference to the multiple capsule, multiple machine case. Accordingly, the client 12 and server machine 11 include a client-side CEE 75 and a server-side CEE 85 loaded in their respective memories.

Figure 3 shows a CEE capsule 70 contained, for example, in a client computer memory 27 (not shown) that includes the CEE 75 and certain of the core CEE components and implementations of objects contained within Implementation Libraries 71. The Implementation Libraries 71 include the client application 79 (or the server application in the case of the server capsule) and client stubs 77 (or server stubs) generated from the IDL specification of the object's interface, as described below. The Implementation Libraries 71 and the CEE 75 interact through the down-calling of

dynamically-accessible routines supplied by the CEE and the up-calling of routines contained in the Implementation Library. The CEE 75 can also receive object calls 82 from other capsules within the same machine and requests 84 from other CEE's. The client-side CEE 75 and the server-side CEE 85 may communicate using any known networking protocol.

Objects implemented in a CEE capsule may be configured or dynamic. Configured objects have their implementation details stored in a repository (such as the MSF Warehouse 85) or in initialization scripts. Given a request for a specific object reference, the CEE 75 starts the appropriate capsule based on this configuration data. The capsule uses the configuration data to determine which Implementation Library to load and which object initialization routine to call. The object initialization routine then creates the object. Dynamic objects are created and destroyed dynamically within the same capsule. Dynamic objects lack repository-stored or scripted configuration information.

III. Compiling and Linking IDL Source Files

Figure 4 shows how IDL source files are compiled and linked into client and server applications that will utilize the method and apparatus of the present invention. First, an IDL source file 101 is prepared containing IDL interface definitions. An IDL compiler 103 compiles the source file 101. The IDL compiler 103 parses the code 101 to produce an intermediate file 105 for storage of the compiled source file. A code generator 112 then parses the PIF file. The code generator 112 generates files in the language of the client and server applications. If the client and server applications are in different languages, different code generators 112 are used. Preferably, the code generator 112 and the IDL compiler 103 are combined in a single application to produce language-specific code. The code generator 112 produces a client stub file 77 containing client stub functions and a server stub file 87 containing definitions for object implementations. The client stub functions include synchronous and asynchronous calls to the CEE 75. The client stub file 77 and the server stub file 87 are compiled by programming language-specific compilers 122, 124 to produce compiled client stub object code and compiled server stub object code. Similarly, a client application 79 and a server application 89 are compiled by programming-

language-specific compilers to produce compiled client application object code and compiled server application object code. The client application 79 and the server application 89 also include a header file 119 generated by the code generator 112. The header file 119 contains common definitions and declarations. Finally, the compiler 121 links the client application object code and the client stub object code to produce an implementation library 71. Similarly, a second compiler links the server application object code server stub object code to produce another implementation library 81.

The code generator 112 produces a Compact IDL Notation ("CIN") and places it in the header file 119. Specifically, the code generator examines all of the compiled interfaces, operations, and operation parameters and defines an identifier (usually the name of the interface) and a character sequence that will be substituted for the identifier in the header file (using, for example, the #define macro in C++). As the code generator 112 parses a source file line-by-line, the code generator stores the sequence of characters in the memory of the computer containing the code generator. Once the entire source file has been parsed, the code generator produces an ASCII descriptor in the header file to be included by the client and server applications. The character sequence in the ASCII descriptor is produced based upon predetermined definitions known to the code generators, as discussed below.

An IDL source file contains one or more interface definitions and data type definitions ("typedefs"). The body of the interface contains several declarations. Constant declarations are used to specify the constants that the interface exports. Type declarations specify the type definitions that the interface exports. Exception declarations specify the exception structures that the interface exports. Attribute declarations specify the associated attributes exported by the interface. Operation declarations specify the operations that the interface exports and the format of each operation, including the name, return data type, parameter types, and exceptions. In a preferred embodiment, the method of the present invention is utilized to describe interface definitions, operation definitions, and individual data type definitions contained in a source file. The method of the present invention may, however, be utilized to describe constant definitions, exception definitions, and module definitions (a definition of interface collections).

IV. Generation of Compact IDL Notation

Figure 5 is a flow chart depicting the generation of a CIN descriptor from an IDL data type, operation, and interface contained in an IDL source file. It will be understood that the steps of Figs. 5-7 are implemented by a CPU of a data processing system executing computer instructions stored in memory. In step 101, the code generator 112 begins with the first line of the IDL source file and determines the data structure, interfaces, or operation described in the source file. If the described data structure is an interface, the code generator 112 follows the directions shown in Figure 7. If the described data structure is an operation within an interface, the generator follows the directions in Figure 6. If the data structure is a data type (or a parameter to an operation as discussed below), the generator generates a single character based upon a table of definitions. It should be noted that different characters may be used than those shown in the charts contained herein. Each chart contains only a preferred ASCII character. Chart A shows a preferred definition table that includes the character strings used to denote the various IDL base types.

IDL Base Type	Representation
Any	A
Boolean	B
Char	C
Double	D
Float	E
Long	F
Long Long	G
Octet	H
Short	I
Unsigned Short	J
Unsigned Long	K

Unsigned Long Long	L
Void	M

Chart A

As shown in Chart A, simple character strings are used to represent base types in a CIN descriptor.

If the data type is a compound type, such as an array or structure (struct type), a series of different steps are followed. In step 111, a character is generated that indicates the start of the compound type. Chart B shows a sample table that includes the character strings used to denote the start of various IDL compound types.

IDL Compound Type	Representation
Array	a
Struct	b
Sequence	c
String	d
Union	e
Union Branch	f

Chart B

The particular representation of each compound type is handled differently according to type.

IDL defines multidimensional, fixed-size arrays for each base type. The array size is fixed at compile time. Arrays are represented in CIN as follows:

a *nr_dimensions size_1. [size_2 , . . . size_n] base type*

The generation of characters for the array is shown in Steps 113, 115, and 117. In this array representation, the character **a** represents the start of the array (as shown in Chart B). The character *nr_dimensions* is a numeral indicating the number of dimensions in the array. The characters *size_1*, *size_2*, *size_n* indicate the size of the array in the first, second, and nth dimension of the array, respectively. For each element of the array, *base-type* is the descriptor for each element. The representation

for the various base types is derived from the original base type table shown in Chart A.

IDL defines user-defined struct types. Each struct is composed of one or more fields of base or compound data types. Structs are represented in CIN as follows:

b *nr_fields field_1 [field_2 . . . field_n]*

The generation of characters to describe a struct is shown in steps 119 and 121. As shown in Chart B, the character **b** indicates the start of the struct. The numeral *nr_fields* indicates the number of fields in the struct. The fields in the struct are described by the descriptors *field_1, field_2, field_n.* Each field is a base or compound type. Base type fields of the struct are described as shown in Chart A. Compound types are described as shown herein (i.e., arrays are described with the start character **a** along with the number of dimensions and the size of each dimension, etc...).

IDL defines sequences of data types. A sequence is a one-dimensional array with two characteristics: a maximum size (fixed at compile time) and a length (determined at run time). Sequences are represented as:

c *nr_occurrences base_type*

The generation of characters for a sequence is shown in steps 123 and 125. In this representation, **c** indicates the start of the sequence as shown in Chart B. The character *nr_occurrences* specifies how many occurrences of the data type are included in the sequence. The number of occurrences is then followed by the actual descriptor, *base_type*, for each occurrence of the data type in the sequence. If the data type is a base type, the appropriate descriptor from Chart A is used. If the sequence consists of compound types, the descriptors are created as described herein. A sequence of sequences is possible.

IDL defines the string type consisting of all possible 8-bit quantities except null. A string is similar to a sequence of chars. Strings are represented in CIN as:

d *size*

The start of the string is indicated by the **d** character. The size of the string is represented by the *size* character generated in step 127.

In IDL, unions are a cross between the "union" of the C programming language and the C "switch" statement. In other words, the union syntax includes a "switch"

statement along with a "case" label indicating the union branches. IDL unions must be discriminated; that is, the union header must specify a typed tag field that determines which union member to use for the current instance of a call. Unions and union branches are represented in CIN as follows:

e *nr_fields* f *label_1 field_1* [f *label_2 field_2 . . . label_n field_n*]

The generation of characters to represent unions and union branches is shown in steps 129, 131, 133 and 135. In this representation, **e** indicates the start of a union and **f** indicates the start of a union branch within the union. The number of fields in the union is specified by the character *nr_fields*. The case label value for each field is indicated by the character *label_1*. If the field is a default, the label is omitted. The descriptor for each field, *field_1*, then follows. The union fields may be either a base or a compound type. Accordingly, the field descriptor for a base type may be generated based upon Chart A. Compound types are generated as described herein.

As seen from the descriptors for base and compound types, the CIN description does not include the identifiers contained in the original IDL source file and a generic descriptor is generated. Thus, even the most complex data structures can easily be represented in string format. The following is a sample structure originally described in IDL:

```
union coordinate_def switch (boolean) {
        case FALSE;
                struct cartesian_def {
                        long x;
                        long y;
                } cartesian;
        case TRUE;
                struct polar_def {
                        unsigned long radius;
                        unsigned long theta;
                } polar;
        };
        typedef sequence<coordinate_def, 100> coordinate_list_def;
```

Using the method of the present invention, the above-described data structure would be represented in CIN as:

"c100+e2+f0+b2+FFf1+b2+KK"

In a preferred embodiment, positive numerals are followed by a plus sign ("+"). Negative numbers are terminated by a negative sign ("-"). While negative numbers

may not occur frequently, their use may be required for certain data types, such as union case labels (i.e., the case discriminator may be a negative number). The CIN descriptor shown above is explained as follows: A data structure consisting of a sequence (c) with a maximum 100 elements (100+), each element consisting of a union (e) with two fields (2+). If the discriminator is zero (FALSE) (f0+) then one variant is a struct (b) containing two fields (2). The first field is a signed long (F). The second field is a signed long (F). If the discriminator is 1 (TRUE) then (f1+) the second variant is a struct containing (b) two fields (2+). The first field is an unsigned long (K). The second field is an unsigned long (K).

If an operation is to be described in CIN, then the method continues at step 151. Figure 6 is a flow chart depicting the steps followed in generating a descriptor for an operation. An operation descriptor is generated in CIN as:

operation_synopsis
operation_id
operation_attribute
nr_params
param_1, param_2 . . . param_n,
nr_exceptions
exception_1, exception_2 . . . exception_n
nr_contexts
context_1, context_2 . . . context_n

In step 151, the code generator generates a unique integer, *operation_synopsis*, that is derived from the string constituting the remainder of the operation's descriptor. The integer is derived by performing, for example, a cyclic redundancy check on the remaining characters in the CIN descriptor. Next, the code generator 112 generates a unique string, *operation_id*, derived from the original IDL name of the operation. Next, in step 155, the code generator generates *operation_attribute*, a character that indicates the attributes (none or "oneway") of the operation. For instance, if the operation has no oneway attribute, the character **A** is generated. If, however, the operation's attribute is oneway, the code generator generates the character **B**. The character *nr_params* is an integer that indicates how many parameters are included in the operation. If the operation has a non-void return type then the first parameter is the result. The *param_1* descriptor includes a character that indicates the direction of the parameter (in, out, inout, or function result) followed by the actual parameter data type. The code generator 112, for example, generates the characters **A**, **B**, **C**, and **D**

for the directions of in, out, inout, and function result, respectively. For the specific parameters, the method returns to step 107 in Figure 5. When the data type of each parameter has been described, the number of exceptions is identified by the integer *nr_exceptions*. The structure description for each exception is then described by returning to step 119, which describes structures. The integer *nr_contexts* indicates the number of context names held by the operation. The names are then generated in strings, *context_1, context_2, context_n*.

The following are two sample operations originally described in IDL:

```
interface Math {
        long Add (in long x, in long y);
        long Subtract (in long x, in long y);
};
```

Using the method of the present invention, the above-described Add operation would be represented in CIN as:

126861413+3+ADDA3+DFAFAF0+0+

The CIN descriptor for the operation is described as follows. The beginning numeral (126861413) is derived from the remainder of the CIN by performing a cyclic redundancy check on the string "3+ADDA3+DFAFAF0+0+". The operation id contains three characters (3+). Those three characters are the string "ADD"--the operation id. All IDL identifiers must be unique and independent of case. Thus, operation id's are capitalized. The operation does not include the oneway attribute (A). The operation includes three "parameters" (3+). Since the function returns a result, the first "parameter" is actually a function result (D). The function result is a signed long (F). The next parameter (actually the first parameter) is an in parameter (A). The parameter is of typed signed long (F). The third parameter is an in parameter (A) of typed signed long (F). There are no exceptions (0+) and no contexts (0+).

Similarly, the CIN descriptor for the Subtract operation would be:

453399302-9+SUBTRACTA3+DFAF0+0+

Interfaces are similarly described using the method of the present invention. Figure 7 shows the generation of interface descriptors. Interfaces are defined as follows:

nr_operations
operation_spec_1
operation_spec_2
operation_spec_n

The integer, *nr_operations* indicates how many operations are contained in the interface. Each operation is then described in *operation_spec_1, operation_spec_2, operation_spec_n* according to the above-described method for generating an operation descriptor. The code generator 112, thus goes to step 151 in Figure 6 to describe each operation.

Using the method of the present invention, the above-described Math interface would be represented in CIN as:

2+126861413+3+ADDA3+DFAFAF0+0+453399302-9+SUBTRACTA3+DFAF0+0+

The interface includes two operations (2+). The operation descriptors for the Add and Subtract operations follow the character indicating the number of operations.

As stated above, the CIN descriptors are contained in a header file that is linked into both the client and server applications. Thus, both the client and the server can make use of the descriptor as each sees fit. The CIN may be used in many ways. For example, a CIN description of a data type may be useful in creating generic functions to pack and unpack structured data types.

CIN descriptions may also be used to compare interfaces quickly. For example, a server application may have a header file containing two interfaces described in CIN. The server may then compare the ASCII string descriptions using known string comparison functions. If the server determines that the CIN descriptions are identical (or similar), the server may implement the operations of both interfaces using common methods in the server application. Thus, the CIN can be used to save time coding multiple methods for different (but similar) interfaces.

Having thus described a preferred embodiment of a method for describing IDL-defined data types, operations, and interfaces, it should be apparent to those skilled in the art that certain advantages of the within system have been achieved. It should also be appreciated that various modifications, adaptations, and alternative embodiments thereof may be made within the scope and spirit of the present invention.

CLAIMS

What is Claimed is:

1. A method for describing an Interface Definition Language-defined data type contained in a source file, the method performed by a data processing system having a memory, and comprising the steps of:

reading the source file; and

generating a string descriptor in the memory, the string identifying the data type.

2. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is a base type, the step of generating a string descriptor in the memory further comprises generating a character indicating the base type.

3. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is an array having n dimensions, each dimension including a base type element, the step of generating a string descriptor in the memory further comprises the steps of:

generating a character indicating a start of the array;

generating a numeral indicating the number of dimensions in the array;

numerals indicating the size of the array in each dimension; and

generating a character indicating the base type for each element in the array.

4. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is a struct having r fields, the step of generating a string descriptor identifying the data type further comprises the steps of:

generating a character indicating a start of the struct;

generating a numeral indicating the number of fields in the struct; and

generating r descriptors indicating a data type for each field in the struct.

5. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is a union having y fields, the step of generating a string descriptor identifying the data type further comprises the steps of:

generating a character indicating a start of the union;

generating a character indicating a start of a union branch;

generating a numeral indicating the number of fields in the union;

generating y numerals indicating a case label value for each field; and

generating y descriptors indicating a data type for each field.

6. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is a sequence, the step of generating a string descriptor identifying the data type further comprises the steps of:

generating a character indicating a start of the sequence;

generating a numeral indicating a number of occurrences of a base type in the sequence; and

generating a descriptor for each occurrence in the sequence.

7. The method for describing an Interface Definition Language-defined data type, as recited in Claim 1, wherein, if the data type is a string, the step of generating a string descriptor identifying the data type further comprises the steps of:

generating a character indicating the string; and

generating a numeral indicating a number of characters in the string.

8. A method for describing an Interface Definition Language-defined operation contained in a source file, the method performed by a data processing system having a memory, and comprising the steps of:

reading the source file; and

generating a string descriptor in the memory, the string identifying the operation.

9. The method for describing an Interface Definition Language-defined operation, wherein the step of generating a string descriptor identifying the operation further comprises the steps of:

generating a character that uniquely identifies the operation;

generating a character string derived from a name of the operation;

generating a character indicating an attribute of the operation;

generating a numeral indicating a number of parameters in the operation;

generating a character indicating a direction for each parameter in the operation;

generating a parameter descriptor indicating a data type for each parameter to the operation;

generating an integer indicating a number of exceptions that can be raised by the operation;

generating a descriptor indicating a structure of each exception that can be raised by the operation;

generating a numeral indicating a number of context names included in the operation; and

generating a descriptor indicating a name of each context included in the operation.

10. The method for describing an Interface Definition Language-defined operation, as recited in Claim 9, wherein, if the data type of a parameter to the operation is an Interface Definition Language base type, the step of generating a parameter descriptor further comprises the step of generating a character indicating the base type.

11. The method for describing an Interface Definition Language-defined operation, as recited in Claim 9, wherein, if the data type of a parameter in the operation is an array having n dimensions, each dimension including an element of a base type, the step of generating a parameter descriptor further comprises the steps of:

generating a character indicating a start of the array;

generating a numeral indicating the number of dimensions in the array;

generating n numerals indicating the size of the array in each dimension; and

generating a character indicating the base type for each element in the array.

12. The method for describing an Interface Definition Language-defined data type, as recited in Claim 9, wherein, if the data type of a parameter in the operation is a struct having r fields, the step of generating a parameter descriptor further comprises the steps of:

generating a character indicating a start of the struct;

generating a numeral indicating the number of fields in the struct; and

generating r data type descriptors indicating a data type for each field in the struct.

13. The method for describing an Interface Definition Language-defined data type, as recited in Claim 9, wherein, if the data type of a parameter of the operation is a union having y fields, the step of generating a parameter descriptor identifying the data type of the parameter further comprises the steps of:

generating a character indicating a start of the union;

generating a character indicating a start of a union branch;

generating a numeral indicating the number of fields in the union;

generating y numerals indicating a case label value for each field; and

generating y data type descriptors indicating a data type for each field.

14. The method for describing an Interface Definition Language-defined data type, as recited in Claim 9, wherein, if the data type of a parameter of the operation is a sequence, the step of generating a parameter descriptor further comprises the steps of:

generating a character indicating a start of the sequence;

generating a numeral indicating a number of occurrences of a base type in the sequence; and

generating an occurrence descriptor for each occurrence in the sequence.

15. The method for describing an Interface Definition Language-defined data type, as recited in Claim 9, wherein, if the data type of a parameter of the operation is a string, the step of generating a parameter descriptor further comprises the steps of:

generating a character indicating the string; and

generating a numeral indicating a number of characters in the string.

16. A method for describing an interface stored in a memory of a computer based upon an Interface Definition Language description of the interface, the method comprising the steps of:

generating an interface descriptor identifying the interface; and

removing a name of an identifier from the interface descriptor.

17. The method for describing an Interface Definition Language-defined interface, as recited in Claim 16, wherein the step of generating an interface descriptor further comprises the steps of:

generating a numeral indicating a number of operations contained within the interface; and

generating an operation descriptor identifying each operation.

18. The method for describing an Interface Definition Language-defined interface, as recited in Claim 17, wherein the step of generating an operation descriptor further comprises the steps of:

generating a character that uniquely identifies the operation;

generating a character string derived from a name of the operation;

generating a character indicating an attribute of the operation;

generating a numeral indicating a number of parameters in the operation;

generating a character indicating a direction for each parameter in the operation;

generating a parameter descriptor indicating a data type for each parameter in the operation;

generating an integer indicating a number of exceptions that can be raised by the operation;

generating an exception descriptor indicating a structure of each exception that can be raised by the operation;

generating a numeral indicating a number of context names included in the operation; and

generating a context descriptor indicating a name of each context included in the operation.

19. A machine readable media embodying instructions for causing the machine to perform a method of describing an Interface Definition Language-defined data type contained in a source file, the method comprising the steps of:

reading the source file; and

generating a string descriptor in the memory, the string identifying the data type.

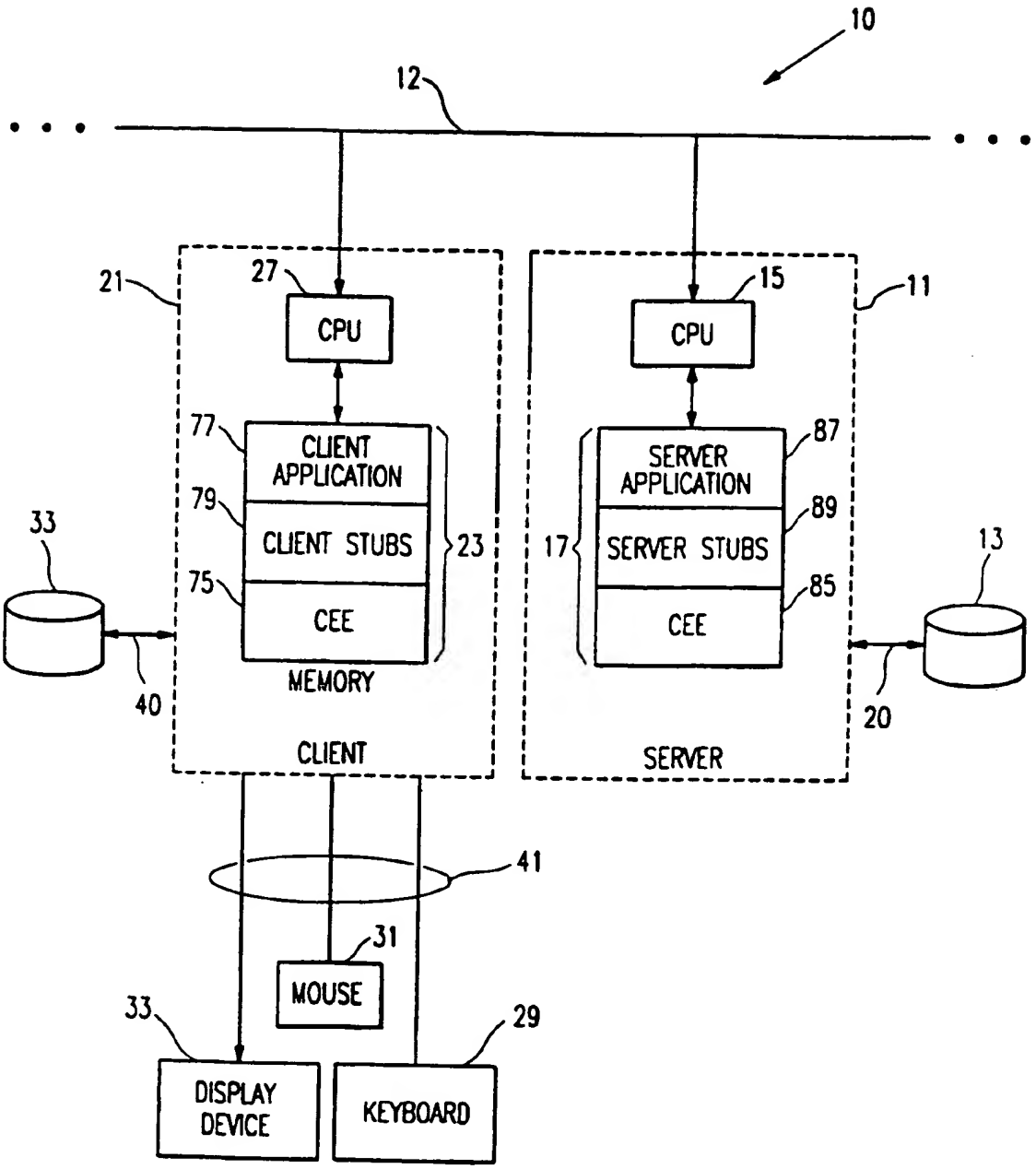

10
12
21
27
CPU
77
CLIENT APPLICATION
79
CLIENT STUBS
75
CEE
23
MEMORY
33
40
CLIENT
15
CPU
11
87
SERVER APPLICATION
89
SERVER STUBS
17
85
CEE
13
20
SERVER
41
31
MOUSE
33
29
DISPLAY DEVICE
KEYBOARD

FIG. 1

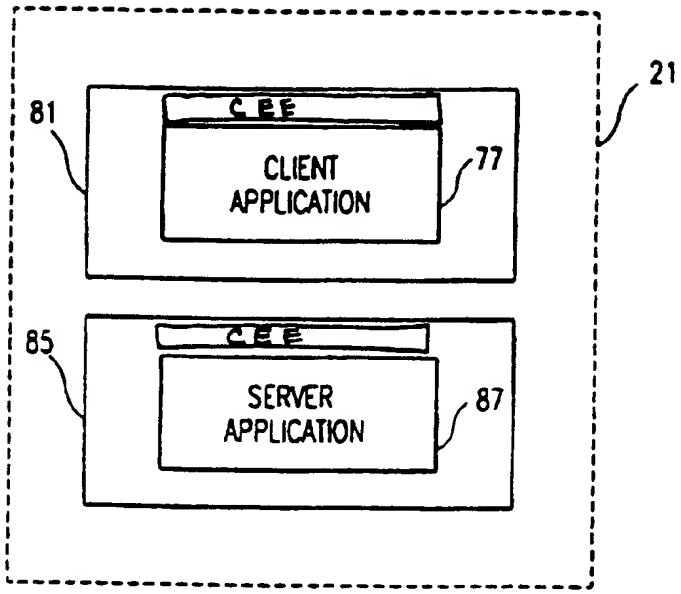

21
81
CEE
CLIENT APPLICATION
77
85
CEE
SERVER APPLICATION
87

FIG. 2A

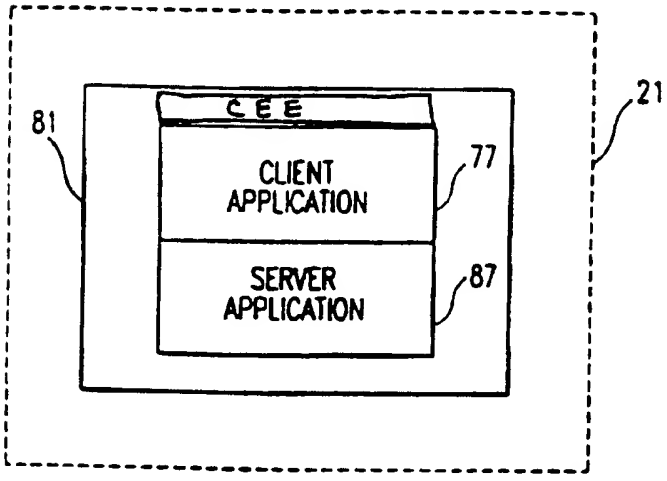

21
81
CEE
CLIENT APPLICATION
77
SERVER APPLICATION
87

FIG. 2B

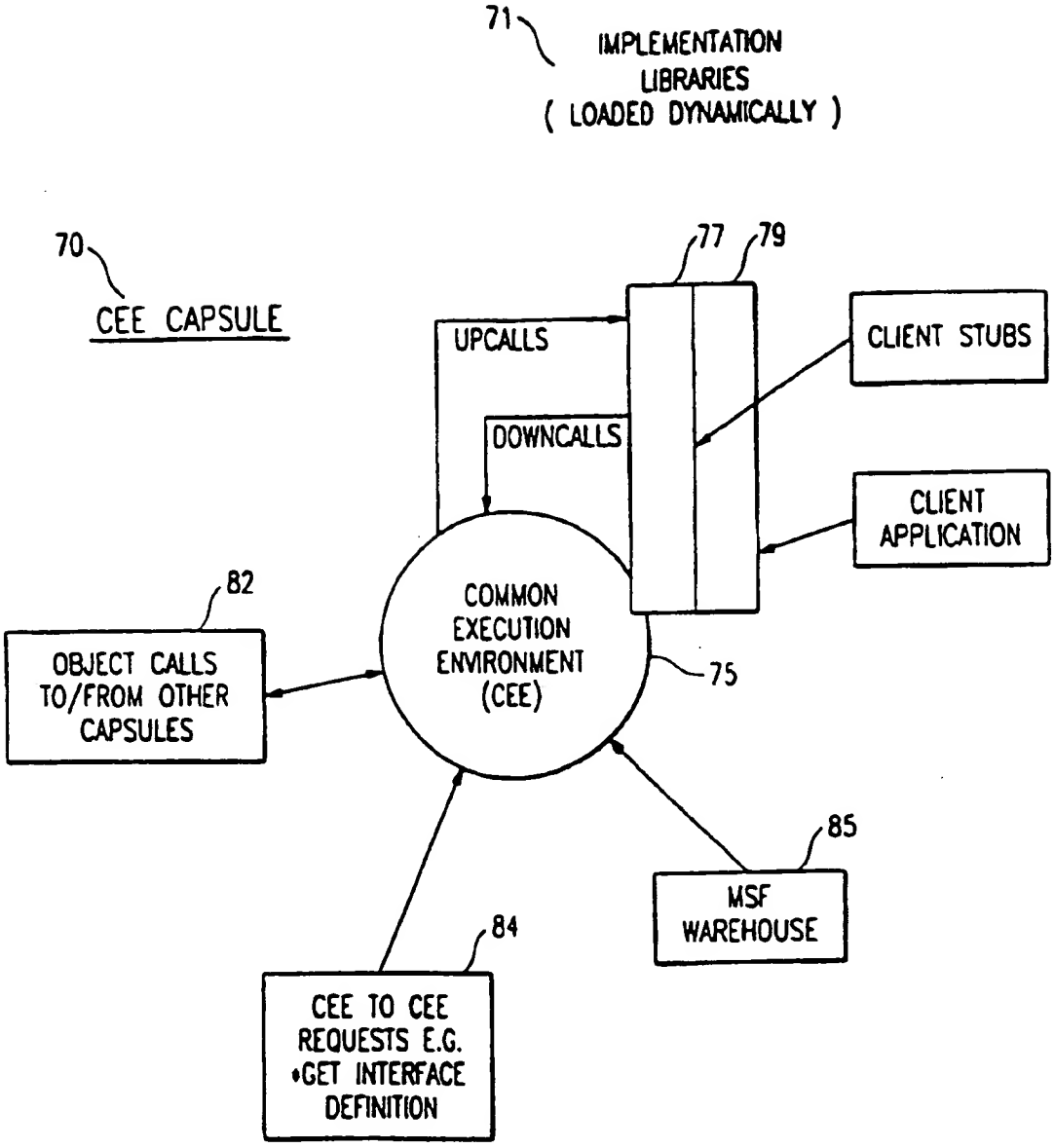

71
IMPLEMENTATION LIBRARIES (LOADED DYNAMICALLY)
70
CEE CAPSULE
77
79
UPCALLS
DOWNCALLS
CLIENT STUBS
CLIENT APPLICATION
82
OBJECT CALLS TO/FROM OTHER CAPSULES
COMMON EXECUTION ENVIRONMENT (CEE)
75
85
MSF WAREHOUSE
84
CEE TO CEE REQUESTS E.G. •GET INTERFACE DEFINITION

FIG. 3

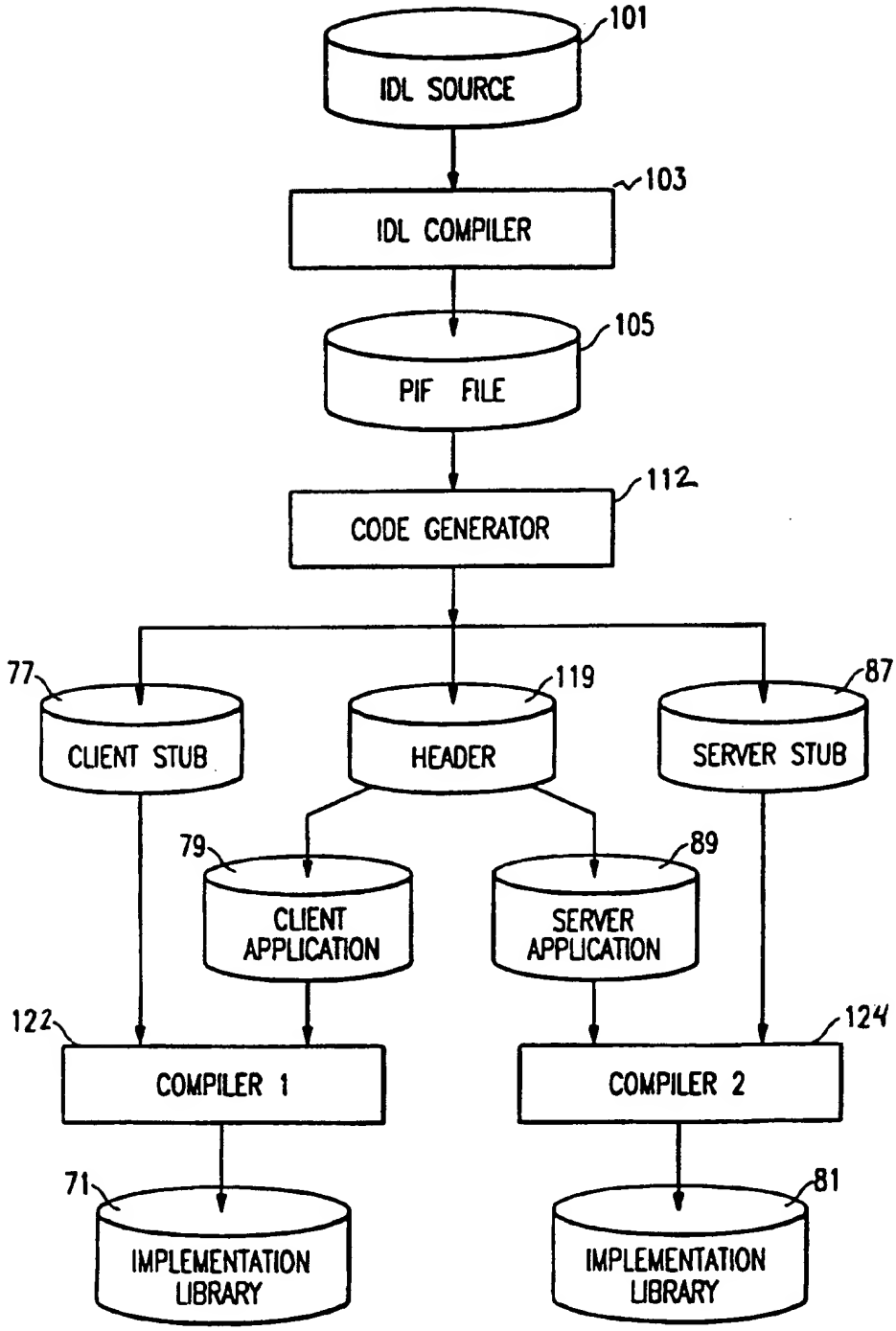

101
IDL SOURCE
103
IDL COMPILER
105
PIF FILE
112
CODE GENERATOR
77
CLIENT STUB
119
HEADER
87
SERVER STUB
79
CLIENT APPLICATION
89
SERVER APPLICATION
122
COMPILER 1
124
COMPILER 2
71
IMPLEMENTATION LIBRARY
81
IMPLEMENTATION LIBRARY

FIG. 4

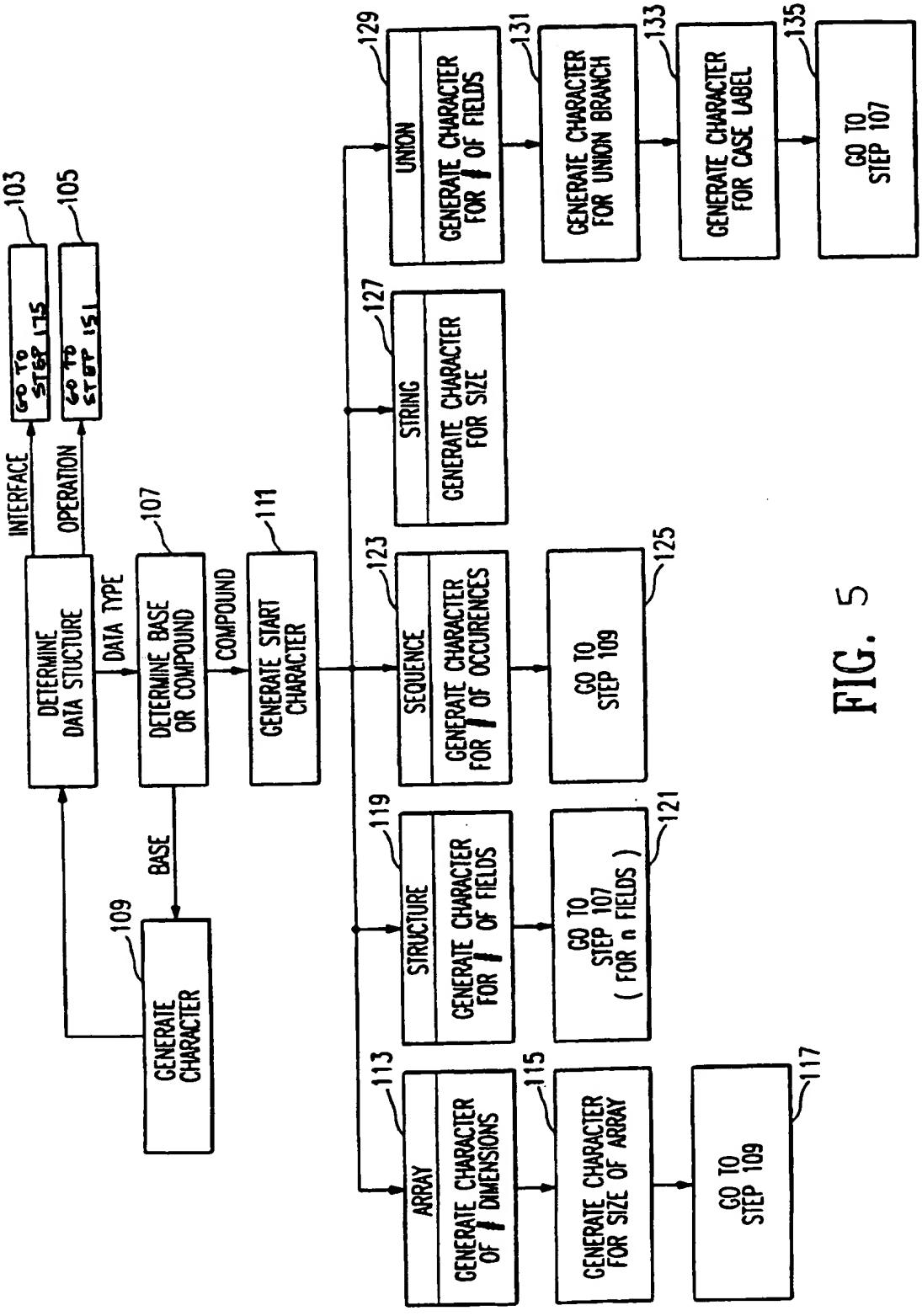

DETERMINE DATA STUCTURE
INTERFACE
OPERATION
DATA TYPE
DETERMINE BASE OR COMPOUND
BASE
COMPOUND
GENERATE CHARACTER
GENERATE START CHARACTER
STRUCTURE
GENERATE CHARACTER FOR # OF FIELDS
ARRAY
GENERATE CHARACTER OF # DIMENSIONS
SEQUENCE
GENERATE CHARACTER FOR # OF OCCURENCES
STRING
GENERATE CHARACTER FOR SIZE
UNION
GENERATE CHARACTER FOR # OF FIELDS
GO TO STEP 109
GO TO STEP 107 (FOR n FIELDS)
GENERATE CHARACTER FOR SIZE OF ARRAY
GO TO STEP 109
GO TO STEP 107
GENERATE CHARACTER FOR UNION BRANCH
GENERATE CHARACTER FOR CASE LABEL
GO TO STEP 107
103
105
107
109
111
113
115
117
119
121
123
125
127
129
131
133
135

FIG. 5

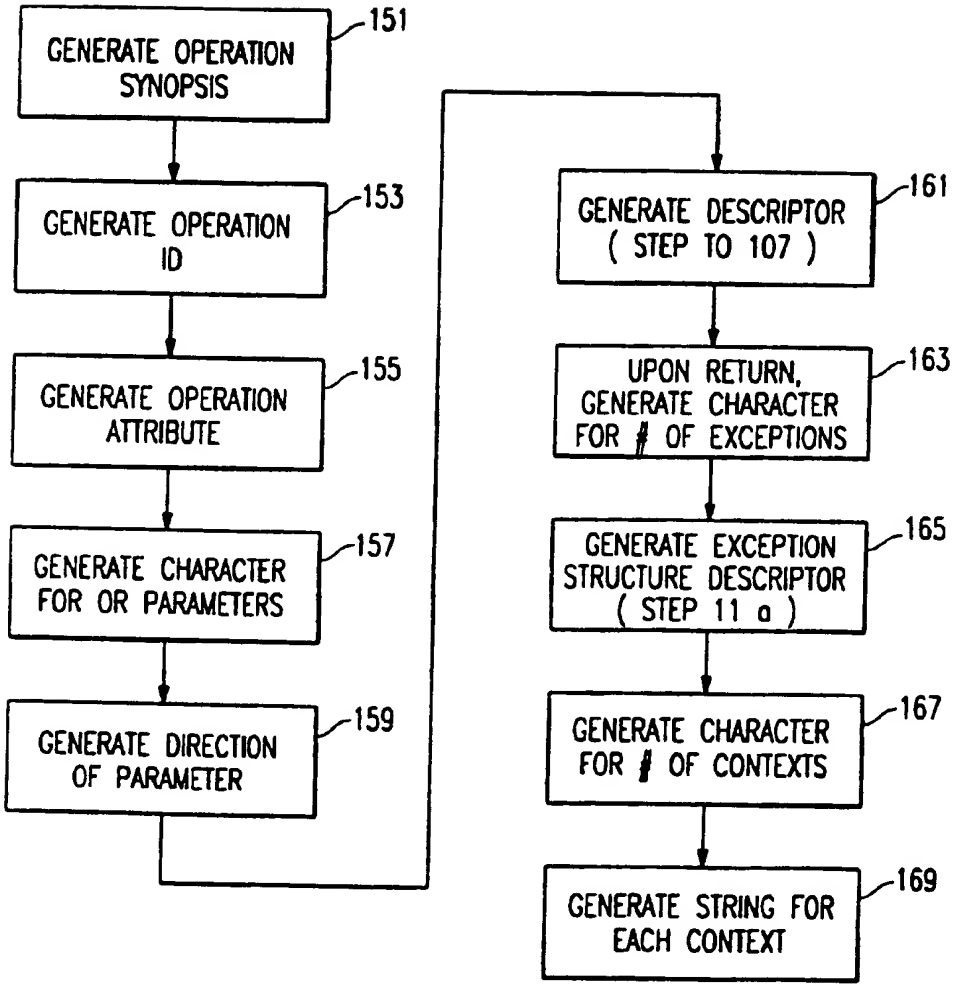

GENERATE OPERATION SYNOPSIS
151
GENERATE OPERATION ID
153
GENERATE OPERATION ATTRIBUTE
155
GENERATE CHARACTER FOR OR PARAMETERS
157
GENERATE DIRECTION OF PARAMETER
159
GENERATE DESCRIPTOR (STEP TO 107)
161
UPON RETURN, GENERATE CHARACTER FOR # OF EXCEPTIONS
163
GENERATE EXCEPTION STRUCTURE DESCRIPTOR (STEP 11 a)
165
GENERATE CHARACTER FOR # OF CONTEXTS
167
GENERATE STRING FOR EACH CONTEXT
169

FIG. 6

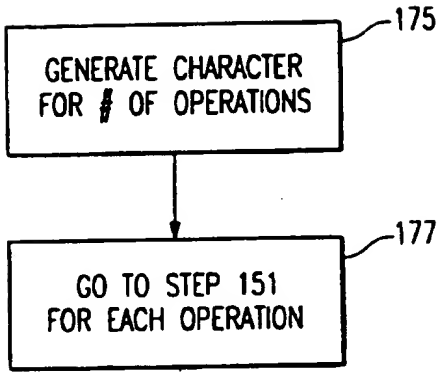

175
GENERATE CHARACTER FOR # OF OPERATIONS
177
GO TO STEP 151 FOR EACH OPERATION

FIG. 7

INTERNATIONAL SEARCH REPORT

	International Application No PCT/US 97/11891

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/45 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	A. VAN WIJNGAARDEN ET AL. (EDS.): "Revised Report on the Algorithmic Language Algol 68" 1976 , SPRINGER-VERLAG , NEW YORK XP002047543 Chapter 7: "Modes and nests" see page 103, line 12 - page 107, line 5 --- -/--	1-19

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
21 November 1997	03.12.97

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Wiltink, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

rnational Application No
PCT/US 97/11891

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	GERALD W. NEUFELD ET AL.: "THE DESIGN AND IMPLEMENTATION OF AN ASN.1-C COMPILER" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. 16, no. 10, October 1990, NEW YORK, US, pages 1209-1220, XP000162480 see abstract see page 1210, right-hand column, line 14 - page 1212, left-hand column, line 9 see page 1214, left-hand column, line 1 - page 1215, right-hand column, line 10 see figures 3-7,10-12; table 1 see Appendix "ASN.1 Structured Type Encoding and Decoding" ---	1-19
A	"DATA STRUCTURES MADE ACCESSIBLE FROM OUTSIDE THE APPLICATION" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 35, no. 2, July 1992, ARMONK, NY, US, pages 255-256, XP000313288 see the whole document ---	1-19
A	J. VAN KATWIJK: "ADDRESSING TYPES AND OBJECTS IN ADA" SOFTWARE PRACTICE & EXPERIENCE, vol. 17, no. 5, May 1987, CHICHESTER, SUSSEX, GB, pages 319-343, XP002030193 see abstract see page 320, line 28 - page 327, line 2; figures 2-6 ---	1-19
A	WO 94 09428 A (MICROSOFT CORP) 28 April 1994 see abstract see page 3, line 9 - page 4, line 9 ---	1,8
A	US 3 886 522 A (BARTON ROBERT S ET AL) 27 May 1975 -----	1,8

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

national Application No
PCT/US 97/11891

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9409428 A	28-04-94	EP 0746815 A	11-12-96
		JP 8502136 T	05-03-96
US 3886522 A	27-05-75	AU 7667174 A	24-06-76
		BE 825392 A	29-05-75
		BE 825393 A	29-05-75
		BE 825394 A	29-05-75
		BE 825395 A	29-05-75
		BE 825396 A	29-05-75
		BR 7500896 A	02-12-75
		CA 1017457 A	13-09-77
		DE 2505842 A	04-09-75
		DK 687074 A	27-10-75
		DK 687174 A	27-10-75
		FR 2262831 A	26-09-75
		GB 1503321 A	08-03-78
		GB 1503322 A	08-03-78
		GB 1503323 A	08-03-78
		GB 1503324 A	08-03-78
		GB 1503325 A	08-03-78
		IN 144139 A	01-04-78
		JP 51098930 A	31-08-76
		JP 50146235 A	22-11-75
		JP 50146236 A	22-11-75
		JP 50146237 A	22-11-75
		JP 50146238 A	22-11-75
		NL 7501332 A	01-09-75
		NL 7501391 A	01-09-75
		SE 413160 B	21-04-80
		SE 7501534 A	29-08-75
		SE 410361 B	08-10-79
		SE 7501535 A	29-08-75
		SE 410360 B	08-10-79
		SE 7501536 A	29-08-75
		SE 410528 B	15-10-79
		SE 7501537 A	29-08-75
		SE 413161 B	21-04-80
		SE 7501538 A	29-08-75
		ZA 7500804 A	25-02-76